Phantom Types and Generalized Algebraic Data Types

Hype for Types

March 2, 2021

Phantom Types

Cup of Tea?

```
fun cupOfTea (wallet : real) =
  (wallet - 3.0, brew ())
```

Cup of Tea?

```
fun cupOfTea (wallet : real) =
  (wallet - 3.0, brew ())

val (wallet', tea) = cupOfTea 100.0
```

Cup of Tea?

```
fun cupOfTea (wallet : real) =
  (wallet - 3.0, brew ())

val (wallet', tea) = cupOfTea 100.0
```

USD or GBP?

Tonnes of Fun

```
val fromGBP : real -> real = fn n => n * 1.27
val cupOfTeaGBP = cupOfTea o fromGBP
```

Fixes?

How can we fix this?

- Vigilance

Fixes?

How can we fix this?

- Vigilance
- Linting/Style Checkers?

Fixes?

How can we fix this?

- Vigilance
- Linting/Style Checkers?
- Types!

First Cut

```
type usd = real
type gbp = real

val fromGBP : gbp -> usd
val cupOfTea : usd -> tea * usd
```

First Cut

```
type usd = real
type gbp = real

val fromGBP : gbp -> usd
val cupOfTea : usd -> tea * usd
```

Oh no!

`real = real`.

Another Try

```
datatype usd = USD of real
datatype gbp = GBP of real

val fromGBP : gbp -> usd
val cupOfTea : usd -> tea * usd
```

Another Try

```
datatype usd = USD of real
datatype gbp = GBP of real

val fromGBP : gbp -> usd
val cupOfTea : usd -> tea * usd
```

Oh no!

How can we add, subtract, etc.? Don't want to write:

```
val add_usd : usd * usd -> usd
val add_gbp : gbp * gbp -> gbp
(* etc. *)
```

Spooky

```
datatype usd = Junk1  (* will never use *)
datatype gbp = Junk2  (* will never use *)

datatype 'a wallet = Wallet of real
(*       ^^  unused type parameter *)

val fromGBP : gbp wallet -> usd wallet
val cupOfTea : usd wallet -> tea * usd wallet
```

Spooky

```
datatype usd = Junk1  (* will never use *)
datatype gbp = Junk2  (* will never use *)

datatype 'a wallet = Wallet of real
(*        ^^  unused type parameter *)

val fromGBP : gbp wallet -> usd wallet
val cupOfTea : usd wallet -> tea * usd wallet

val + : 'a wallet * 'a wallet -> 'a wallet
val - : 'a wallet * 'a wallet -> 'a wallet
(* etc. *)
```

Spooky

```
datatype usd = Junk1  (* will never use *)
datatype gbp = Junk2  (* will never use *)

datatype 'a wallet = Wallet of real
(*       ^^  unused type parameter *)

val fromGBP : gbp wallet -> usd wallet
val cupOfTea : usd wallet -> tea * usd wallet

val + : 'a wallet * 'a wallet -> 'a wallet
val - : 'a wallet * 'a wallet -> 'a wallet
(* etc. *)
```

Phantom Type

Since the parameter `'a` doesn't appear in the definition of `wallet`, we call `wallet` a *phantom type*.

Lo Hicimos!

How can we use it?

```
val ronWallet : usd wallet = Wallet 50.0
val steveWallet : gbp wallet = Wallet 42.0

val (ronWallet', tea) =
  cupOfTea ronWallet

val (steveWallet', tea) =
  cupOfTea steveWallet
(* TYPE ERROR *)

val (steveWallet', tea) =
  cupOfTea (fromGBP steveWallet)
```

Pushing it Further

```
datatype ('a,'b) exchange = Exchange of real
val convert :
  ('a,'b) exchange
  -> 'a wallet -> 'b wallet =
  fn Exchange rate =>
    fn Wallet n => Wallet (rate * n)

val ex : (gbp,usd) exchange = Exchange 1.27
val fromGBP = convert ex
(* : gbp wallet -> usd wallet *)
```

Pushing it Further

```
datatype ('a,'b) exchange = Exchange of real
val convert :
  ('a,'b) exchange
  -> 'a wallet -> 'b wallet =
  fn Exchange rate =>
    fn Wallet n => Wallet (rate * n)

val ex : (gbp,usd) exchange = Exchange 1.27
val fromGBP = convert ex
(* : gbp wallet -> usd wallet *)

datatype cad = Junk3

val cadExchange : (usd,cad) exchange = Exchange 1.33
val fromUsd = convert cadExchange
```

Key Point

Key Point

Type parameters can be "compile-time only"! They need not be used at runtime.

We can use this to help our compiler check extra invariants.

GADTs

Arrays

```
signature ARRAY =
  sig
    type 'a t
    val fromList : 'a list -> ' a t
    val fromInt : int -> bool t
    val get : int -> 'a t -> 'a
  end
```

First Try

```
structure Array :> ARRAY =
  sig
    datatype 'a t =
      List of 'a list
    | Int of int
    val fromList = List
    val fromInt = Int
    (* get? *)
```

First Try

```
structure Array :> ARRAY =
  sig
    datatype 'a t =
      List of 'a list
    | Int of int
    val fromList = List
    val fromInt = Int
    (* get? *)

    fun get i = fn
      List xs => List.sub (xs,i)
    | Int n   => ((n >> i) & 1) > 0
  end
```

Oh No

Type Error!

`get : bool array -> bool`

But we said it would have type `'a array -> 'a`.

We have to give back an `'a` in the `Int` branch, but cannot.

The only way to use the `Int` constructor is through `fromInt`, which produces a `bool array`.

But the compiler doesn't know that :(

Hmm

What if our compiler knew that if we match on `Int`, `'a` must be `bool`?

Generalizing ADTs

An alternative syntax for ADTs

```
datatype 'a option =
  SOME : 'a -> 'a option
| NONE : 'a option

datatype 'a list =
  Nil  : 'a list
| ::   : 'a * 'a list -> 'a list

datatype 'a array =
  List : 'a list -> 'a array
| Int  : int -> 'a array
```

Generalizing ADTs

Just change the return type for Int!

```
datatype 'a array =
  List : 'a list -> 'a array
| Int  : int -> bool array
```

Nice

get typechecks now!

```
fun get i = fn
  List xs => List.sub (xs,i)
| Int n   => ((n >> i) & 1) > 0
```

In the `Int` arm of the case, `'a` gets *refined* to `bool`.

- The compiler knows that `Int : int -> bool array`
- So `Int n : bool array`
- So it must be that `'a = bool`

Exhaustiveness

```
val toString : char array -> string = fn
  List xs => String.implode xs
| Int n    => ???
```

Exhaustiveness

```
val toString : char array -> string = fn
  List xs => String.implode xs
```

There's no way to create a `char array` with the `Int` constructor!

This pattern match is actually exhaustive.

List Frustrations

```
val head : 'a list -> 'a = fn
  x::xs => x
| []    => raise Fail "oop"

val zip : 'a list * 'b list -> ('a * 'b) list = fn
  ([],[])         => []
| (x::xs,y::ys) => (x,y)::zip (xs,ys)
| _               => raise Fail "oop"
```

Can we fix it?

Want to statically detect calling head on empty lists and zip on lists on non-equal length

Thoughts?

Can we fix it?

Want to statically detect calling head on empty lists and zip on lists on non-equal length

Thoughts?

What if the type checker knew how long a list was?

Length Indexed Lists : First Try

What we'd really like:

```
datatype ('a,'len) list =
  Nil  : ('a,0) list
| :: : 'a * ('a,'len) list -> ('a,'len + 1) list
```

But 0 and 1 aren't types :(

Length Indexed Lists : First Try

What we'd really like:

```
datatype ('a,'len) list =
  Nil  : ('a,0) list
| :: : 'a * ('a,'len) list -> ('a,'len + 1) list
```

But 0 and 1 aren't types :(
Workarounds?

Type Level Naturals

We need to encode the natural numbers into our type system!

Type Level Naturals

We need to encode the natural numbers into our type system!

```
(* Constructors could be anything *)
(* We just need a new type *)
datatype z = Junk of void
```

Type Level Naturals

We need to encode the natural numbers into our type system!

```
(* Constructors could be anything *)
(* We just need a new type *)
datatype z = Junk of void

type 'n s = Junk of void (* same deal *)
```

Now we have a *type* that corresponds to each nat!

Length Indexed Lists : Second Try

```
datatype ('a,'len) list =
  Nil : ('a,z) list
| ::  : 'a * ('a,'len) list -> ('a,'len s) list
```

List Frustrations Alleviated

Can we express the desired constraints on `head` and `zip` now?

List Frustrations Alleviated

Can we express the desired constraints on `head` and `zip` now?

```
val head : ('a,'n s) list -> 'a = fn
  (x::xs) => x
```

List Frustrations Alleviated

Can we express the desired constraints on `head` and `zip` now?

```
val head : ('a,'n s) list -> 'a = fn
  (x::xs) => x

val zip : ('a,'n) list * ('b, 'n) list ->
          ('a * 'b,'n) list = fn
  ([],[])       => []
| (x::xs,y::ys) => (x,y)::zip (xs,ys)
```

All patterns are fully exhaustive!

How great is this really?

```
val append : ('a,'n) list * ('a,'m) list ->
             ('a,???) list

val filter : ('a -> bool) -> ('a,'n) list ->
             ('a, ???) list
```

We need a much more powerful type system to express the types of functions that alter list lengths in complex ways. We'll get there!

Pushing Type Nats Further

Any other data structures where statically tracking a number could prove useful?

Pushing Type Nats Further

Any other data structures where statically tracking a number could prove useful?

Red-Black Tress!

Pushing Type Nats Further

Any other data structures where statically tracking a number could prove useful?

Red-Black Tress!

If we encode our invariants at the type level, we can guarantee any functions on red-black trees cannot break them

- All nodes are either red or black
- The empty tree is black
- All leaves are black
- Red nodes have black children
- Any path from a node to one of its descendant leaves has the same number of black nodes

Red-Black Trees

```
datatype red = Junk of void
datatype black = Junk of void
datatype ('a,'color,'n) tree =
  Empty : ('a,black,z) tree

| Red    : ('a,black,'n) tree *
           ('a,black,'n) tree *
           'a ->
           ('a,red,'n) tree

| Black : ('a,'c1,'n) tree *
          ('a,'c2,'n) tree *
          'a ->
          ('a,black,'n s) tree
```